More specifically, Sections 2 and 3 describe the conceptual foundations of the concept of subsetability: "nearly decomposable hierarchies" and "increasingly complex microworlds." Section 4 defines subsetability, and Section 5 shows how the concept applies to PL/Es. Section 6 claims that a PL/E's subsetability positively affects its learnability and teachability, and describes some research supporting that claim. Section 7 presents arguments that subsetability may be a new cognitive dimension of notational systems. The final section offers a summary and some potential ramifications of this on-going research.

2 Nearly Decomposable Hierarchies

Simon eloquently explains that many natural systems are *nearly decomposable hierarchies*. He argues that their decomposability facilitates our learning and teaching of them. "The fact then that many complex systems have a nearly decomposable, hierarchic structure is a major facilitating factor enabling us to understand, describe, and even 'see' such systems and their parts. Or perhaps the proposition should be put the other way around. If there are important systems in the world that are complex without being hierarchic, they may to a considerable extent escape our observation and understanding" [2]. By implication, the fact that many natural – and artificial – systems are only "nearly" decomposable *inhibits* our learning and teaching of them.

3 Increasingly Complex Microworlds

Related to Simon's concept of system decomposition is the *microworld* concept. The term *microworld* was popularized by Papert in his classic "Mindstorms" book. A microworld is "an incubator for knowledge, a 'place' ... where certain kinds or mathematical thinking could hatch and grow with particular ease" [3]. As the name implies, a microworld is a "small world," a subset of some "world" of skills that a student is attempting to learn.

According to Burton, Brown, and Fischer, "A microworld is created by manipulating three elements: the *equipment* used in executing the skill, the *physical setting* in which the skill is examined, and the *task specification* for the given equipment and physical setting" [4]. For example, in the world of skiing, equipment includes skis of various lengths, bindings, poles, etc. Physical settings include mild slopes, steep slopes, combinations of downward and upward slopes, etc. Task specifications include snowplowing, stopping, changing direction, etc. A teacher should construct a microworld so it provides:

- "The right entry points into an environment, making it easier to get started on a subskill.
- An environment in which the student feels safe, allowing him to focus his attention on learning skills.
- Intermediate goals or challenges that are, and seem to be, attainable.

- Practice of the important subskills in isolation, allowing the common 'bugs' to occur one at a time instead of in bunches" [4].

Burton et al elaborate upon the microworld concept by suggesting that a teacher should define a linear sequence of microworlds to guide the student toward understanding of the world at hand. The first microworld in the sequence should constrain the world substantially, and each subsequent microworld should remove some of the constraints imposed by its predecessor. Thus the student learns about the world by traversing a linear sequence of *increasingly complex microworlds* (ICM).

4 Subsetability

Inspired by Simon's concept of nearly decomposable hierarchy, we suggest that the ICM concept can be extended beyond the linear topology that Burton et al describe. Specifically, we suggest that many "worlds" can and should be decomposed into a *hierarchy* (more properly, a *directed acyclic graph*, alias *lattice*) of microworlds, of the form illustrated by Figure 1.

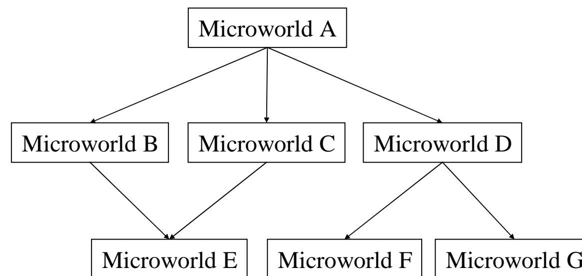

Fig. 1. A lattice of microworlds

In the lattice, each node denotes a microworld, and each directed edge denotes a learning dependency: an edge from node X to node Y (for any X and Y) indicates that the student must become comfortable with the skills illustrated by microworld X before moving to microworld Y. In other words, microworld X is prerequisite to microworld Y. Part of the teacher's job is to present the student with a linear sequence of microworlds that respects the learning dependency relationships defined by the lattice.

As with Burton et al's linear model, we suggest that each microworld in the lattice should be small, thus allowing the student to focus on a few new skills. More precisely, the "root" microworld(s) should constrain the world substantially, and each "child" microworld should remove only a few of the constraints imposed by its parent microworld(s).

We define *subsetability* as the extent to which a world can be decomposed into such a lattice of microworlds.

5 Subsetability of Programming Languages and Environments

In our experience, the concept of subsetability applies to the learning and teaching of PL/Es. In the world of PL/Es, the equipment consists of the programming language: C, C++, Java, Scheme, etc. The physical setting consists of the programming environment: text editor and command-line interface, Eclipse, Microsoft Visual C++, Borland JBuilder, etc. The task specifications consist of programming projects. Examples of reasonable microworlds for many PL/Es might be entitled hello-world, read-compute-write, transfer-of-control, functional-decomposition, arrays, files, classes-and-objects, etc.

To illustrate the applicability of subsetability to PL/Es, we note that many popular PL/Es are not particularly subsetable, especially near the "roots" of their microworld lattices. Consider these examples:

5.1 Example 1: "Hello-World" in C

Consider the hello-world microworld – arguably the microworld at the root of the lattice of many PL/Es. That microworld has a single task specification: command the computer to print "hello, world." Suppose the equipment is the C programming language, and the physical setting consists of an ordinary text editor and command-line interface.

To implement the task, the student must use the editor to create, and the command-line interface to prepare, this code:

```
#include <stdio.h>
int main(void)
{
   printf("hello, world\n");
   return 0;
}
```

Note how many language features the student must know to create that code: preprocessor directives, header files, function declarations, function definitions, function return types, the main() function, parameters, "void" as a parameter specification, statements, function calls, the printf() function, strings, the newline character, the "return" statement, function return values, and the "0 as successful termination" convention. Thus note the enormous size of the equipment subset corresponding to that microworld. In that sense, we claim that the C/text-editor PL/E world is not particularly subsetable, at least at the root of its microworld lattice. The hello-world microworld has an equipment set that is not "micro."

5.2 Example 2: "Hello-World" in Java

The Java programming language is even worse with respect to subsetability at the root of its microworld lattice. Consider the same hello-world microworld implemented using Java as the equipment and a text editor and command-line interface as the physical setting. To implement the task, the student must create this code:

```
class Hello
{
   public static void main(String[] args)
   {
      System.out.println("hello, world");
   }
}
```

Again note how many language features the student must know: classes, method definitions, the main() method, public methods, static methods, method return types, "void" as a method return type, the String class, arrays, parameters, statements, the System class, static fields, the public static "out" field within the System class, the println() method within the PrintStream class, and String literals. The equipment set for the hello-world microworld has enormous cardinality. Thus we claim that the Java/text-editor PL/E is not particularly subsetable, at least at the root of its microworld lattice.

5.3 Example 3: "Read-Compute-Write" in C

Now consider the read-compute-write microworld – arguably a microworld *near* the root of the microworld lattice of many programming languages. A typical task specification in that microworld might be "read an integer, and compute and write its square." Suppose the equipment is the C programming language, and the physical setting is a text editor and command-line interface. The student must create this code:

```
#include <stdio.h>
int main(void)
{
   int i;
   printf("Enter an integer:\n");
   scanf("%d", &i);
   printf("The integer squared is %d\n", i * i);
   return 0;
}
```

Note that the student must learn several new language features beyond those used in the hello-world microworld: variables, the scanf() function, scanf() conversion specifications, printf() conversion specifications, and the multiplication operator. Most problematic is the need to use the "address of" operator in the call to scanf() – an operator whose meaning the student cannot adequately understand until he/she becomes comfortable with the, say, arrays-and-pointers microworld, which appears much lower in the lattice. In that sense, we claim that the C/text-editor PL/E is not

particularly subsetable near the root of its microworld lattice. The read-compute-write microworld has an equipment set that is not "micro."

5.4 Example 4: "Read-Compute-Write" in Java

Finally, consider the read-compute-write microworld and its "read an integer, and compute and write its square" task in Java using a text editor and command-line interface. The student must use the editor to create this code:

```
class Square
{
   public static void main(String[] args)
   {
      int i;
      System.out.println("Enter an integer:");
      i = NonStandardClass.readInt();
      System.out.print("The integer squared is ");
      System.out.println(i * i);
   }
}
```

Again note that the student must learn a few language features beyond those in the hello-world microworld: variables, the print() method within the PrintStream class, and the multiplication operator. More significantly, the Java language contains no standard class for handling keyboard input. So the teacher must introduce some non-standard class, accompanied by some rather awkward baggage: instructions about how to place that class in the CLASSPATH, perhaps by copying the class definition into the current directory, explanations about why that class does not appear in the Java documentation or textbook, etc. Thus we claim that the Java/text-editor PL/E is not particularly subsetable near the root of its microworld lattice.

5.5 The Interplay of Programming Language and Environment

In fact, for the C/text-editor and Java/text-editor PL/Es, the ICM approach is not feasible in its proper form. Instead the teacher must formulate compromised microworlds consisting of proper physical settings (programming environment), proper task specifications (programming project), and "improper" equipment sets (programming language features) containing multiple forward references to features that more properly belong to "child" microworlds. The teacher must ask the students blindly to accept the forward references "on faith" or "as part of a fixed template," with the promise of covering those advanced features later.

Note, however, that the choice of physical setting (programming environment) can affect a world's subsetability dramatically. For example, the ProfessorJ programming environment [5] implements a read-eval-print loop (REPL) that allows students to instantiate new objects of existing classes, and then send them messages, without defining a main() method. Thus the Java/ProfessorJ PL/E has a substantially different

microworld lattice than the Java/text-editor PL/E does. That microworld might be substantially more subsetable.

6 PL/E Subsetability, Learnability, and Teachability

We believe that the "nearly decomposable hierarachy" and "increasingly complex microworlds" theoretical arguments strongly suggest that a PL/E's subsetability positively affects its learnability and teachability. Can the effect be shown empirically? The following sections describe several efforts to develop or evaluate restricted languages for novice programmers.

6.1 Language Levels

Findler and his colleagues developed DrScheme, a programming environment for Scheme designed to support students in introductory courses [6, 7]. While the environment includes many tools to aid students, the most interesting aspect of DrScheme is the implementation of language levels. A language level provides the programming language features needed to teach a certain set of constructs, but it hides language features that are unnecessary and may interfere with the current pedagogical goals. For example, a beginning student might use an identifier without realizing that it is a keyword. This would result in an error message that is inexplicable to the student, at once frustrating and slowing the progress of the individual.

DrScheme implements four language levels, with each successive level extending the previous level. The Beginner Level includes definitions, conditionals and a large number of functional primitives. The Intermediate Level provides structure definitions and local binding constructs. The Advanced Level adds variable assignments, data mutations, and implicit and explicit sequencing. Finally, the highest level, referred to as Full, corresponds to the full Scheme language. DrScheme has been used in introductory programming courses but no empirical evaluation has been done. Another similar pedagogical environment, ProfessorJ, implements language levels for Java, but like DrScheme has not been evaluated rigorously in the classroom [5].

6.2 Mini-Languages, Sub-languages, and the Incremental Approach

Mini-languages and sub-languages are two approaches for helping beginners learn to program. As described by Brusilovsky, mini-languages are small, simple languages meant to support the earliest experiences in programming [8, 9]. Mini-languages often have physical actors, such as a turtle, that the learner controls by issuing simple commands or by writing programs in the case of executing more complex actions. Mini-languages also can have virtual actors that the learner controls via programming [10]. One advantage of mini-languages is that the language itself is small and simple,

and in addition the learner is not exposed to a complex environment. Another advantage is that the actions of the actors are visible, allowing learners to understand the effects of their programs. The syntax and semantics of mini-languages are usually not tied directly to the syntax and semantics of a professional programming language. This allows students to learn principals of problem solving through programming without the overhead of a full programming language [8, 9].

Sub-languages are similar to mini-languages in their goal of introducing beginners to programming in a way that protects them from the complexity of a full-blown programming language. However, sub-languages differ from mini-languages because they are "starting subsets" of a full language rather than miniature languages with their own unique syntax and semantics [8]. A sub-language extends a programming language by providing additional commands used in combination with standard control structures of the programming language. For example, including turtle graphic commands in a standard programming language is an extension to aid learners. The sub-language approach may be better than mini-languages for students who plan to go on to learn the full language because the transfer to the full language will be smoother. In effect, moving to the full language is equivalent to the concept of fading scaffolds as the student gains mastery [11, 12]

Finally, the incremental approach to language learning (also referred to as subsetting) is also meant for learning introductory programming in a manner that will help the learner to migrate to the full language. The difference between the incremental approach and the sub-language approach is that in the incremental approach the student is taught a *sequence* of language subsets, leading toward the full programming language [8]. There are no special purpose commands and features that extend the standard language for novices. Instead the learner moves from one subset to the next in a staged manner, while keeping all the features of the previous subset(s). Each subset is defined and can be learned independent of other subsequent subsets [8]. DrScheme and ProfessorJ are examples of the incremental approach.

6.3 Programming Environment Subsets

As mentioned previously, subsetting has been developed and used as a pedagogical strategy in DrScheme and ProfessorJ, but there have been no empirical evaluations of them. However, one empirical evaluation of subsetting has been reported in the literature [13]. The setting of the evaluation was a first programming course using Java over a semester of 13 weeks. Two experimental groups used Java with an environment specially designed for simplicity: one group used subsetting while the other used the full Java language. The two groups worked in the simple environment for nine weeks and then both groups transferred to the professional Java.net environment (including the full Java language) for the last four weeks of the semester. The results for the first nine weeks indicated that there was no significant difference in performance between the group that used subsetting and the group that used the full Java language. Furthermore, when the two groups migrated to Java.net there was no significant difference in their assignment scores. While these results do not favor subsetting, more empirical study is warranted. This is a single study that needs to be repli-

cated, perhaps with methodological changes. In particular, the study appears to have faced the dilemma of ecological validity versus experimental control.

7 Subsetability as a New Cognitive Dimension

The rest of this paper addresses the issue of whether subsetability might be considered a new cognitive dimension (CD) of notational systems. In that regard, Blackwell [14] provides a road map. He lists six criteria that any proposed new cognitive dimension should satisfy: applicability, polarity, object of description, orthogonality, granularity, and effect of manipulation. Each of the following subsections considers subsetability with respect to one of those criteria.

7.1 Applicability

"One of the desirable properties of a CD is that it should make sense to talk about it in a wide range of different situations. This has not always been achieved with the current set of dimensions" [14].

We believe that subsetability applies to notational systems in addition to PL/Es. Consider alarm clocks – notational systems far removed from PL/Es. Although using alarm clocks is much simpler than using PL/Es, we submit that alarm clocks, to varying degrees, suggest simple lattices of microworlds: set-and-read-the-time, set-and-activate-and-deactivate-the-alarm, set-the-snooze-feature, etc. Thus we believe that subsetability applies to alarm clocks.

Consider video cassette recorders. We submit that VCRs suggest lattices of microworlds: set-time, playback-program, record-program-now, record-program-later, record-multiple-programs-later, record-one-program-while-watching-another, duplicate-tape, etc. Indeed we suggest that the infamous difficulty of learning to use VCRs may be related to their non-subsetability. For example, to set the time (a task within a simple microworld), many VCRs require the user to learn how to traverse an elaborate hierarchical function menu (ideally, a task within more complex microworlds).

Indeed, in accord with the theoretical arguments described previously, we believe that the concept of subsetability applies to any "system" (Simon) or "world" (Burton, Brown, and Fischer), and thus any notational system, that users must learn.

7.2 Polarity

"As CDs are not supposed to be either good or bad ..., they should have interesting properties in both directions – i.e. both when present and absent" [14].

As noted previously, we believe that subsetability has a beneficial effect upon learnability. But it is also possible that subsetability could have a detrimental effect upon learnability, perhaps depending upon the nature of the notational system or the

nature of the learner. For example, for some notational systems and learners, subsetability might imply faster/deeper learning via smooth learning paths. For other notational systems or learners, non-subsetability might imply faster/deeper learning because "forward references" attune the student to the existence of advanced features, and motivate the student to learn those features.

7.3 Object of Description

"There is an outstanding question regarding what it is that the dimensions are supposed to describe. Some possible options for suitable objects of description (no doubt not a complete list) are: (i) structural properties of the information within the notation/device, (ii) the external representation of that structure (iii) the semantics of that information, and (iv) the relationship between the notated information and domain-level concepts – some of which are inevitably not notated... Regarding the definition of a criterion for new dimensions … whichever subset of (i)-(iv) (or more) we choose, the proposed dimension should describe something that falls within that subset" [14].

Subsetability describes structural properties of the notational system itself (i.e. PL/Es), rather than structural properties of artifacts built using the notational system (i.e. computer programs). Given that distinction, we claim that subsetability falls within option (ii), which we paraphrase as "the external representation of the structural properties of the information within the notation." Frankly, we are somewhat unclear about the interpretation of that option, so we make the claim only tentatively.

7.4 Orthogonality

"Most important, the term 'dimension' was chosen to imply that these are mutually orthogonal – they all describe different directions within the design space. Furthermore, it is hoped that the trade-off relationships between them might be similar to those of the Ideal Gas Law – so that it is probably not possible to design a notation system that achieves specific values on any two dimensions, without having the value of a third imposed by necessary constraints. But these notions of orthogonality are intuitive rather than exact, and they are described in this way mainly so that designers recognise the nature of the constraints on their design... mutual orthogonality can only really be tested at present via a qualitative approach – going through all current dimensions, and checking to see whether any of them might describe the same phenomenon as that described by the proposed new dimension... It is also necessary to be aware that the new dimension might simply be the obverse case of an existing dimension" [14].

The most authoritative list of cognitive dimensions is given in the cognitive dimensions tutorial [1]. We indeed did go through all dimensions described in that paper, checking to see whether any of them might describe the same phenomenon as that described by subsetability.

In our opinion, of all the cognitive dimensions described by that paper, *abstraction* is the closest to subsetability. "An abstraction is a class of entities, or a grouping of elements to be treated as one entity, either to lower the viscosity or to make the notation more like the user's conceptual structure" [1].

At first glance, abstraction and subsetability seem quite similar, as illustrated by these quotes:

- "Systems that enforce abstraction will be difficult to learn" [1]. Similarly, we believe that systems that are not subsetable will be difficult to learn.
- "The abstraction barrier is determined by the minimum number of new abstractions that must be mastered before using the system" [1]. Similarly, we believe that systems that are not subsetable present the user with a "subsetability barrier" that requires the user to learn many new features before using the system.
- "Programming languages often use many abstractions (procedures, data structures, modules, etc.) although spreadsheets use rather few" [1]. Similarly, the concept of programming language subsetability is built upon programming language "features," some of which are procedures, data structures, and modules. And programming language indeed do contain more features than spreadsheets do.

However, consider these quotes:

- "The characteristic of an abstraction from our point of view here is that it changes the notation" [1]. But subsetability describes a notational system "out of the box," and is unrelated to changes made by the user.
- Abstractions "must be maintained... Creating and editing them takes time and effort and has potential for mistakes" [1]. But, again, subsetability is unrelated to user changes to the notational system.

So, clearly, abstraction and subsetability are different concepts. The distinction is related to the previous "object of description" analysis. Whereas subsetability is a property of a notational system (e.g. a PL/E), we believe that abstraction is primarily a property of the artifacts created using that notational system (e.g. computer programs). Subsetability comments on the learnability of PL/E entities such as procedures, data structures, classes, and modules, all of which are part of the PL/E's baseline syntax and semantics. Abstraction comments on the learnability of computer program entities such as procedure A, data structure B, class C, and module D, all of which are defined by the user.

7.5 Granularity

"The CDs seem to describe activities at a reasonably consistent level of granularity. It is probably a good thing that they should continue to describe phenomena at a similar scale. They do not directly describe large cognitive tasks (design a system, write a play), but the structural constituents of those tasks. They also tend not to describe low-level perceptual processes... Some things that are too low a level of granularity might include Gestalt phenomena, or observations related to individual motions (e.g.

selection target size, as analysed by Fitts' law). If they were to be characterised using GOMS analysis (which they are not going to be …), we might say that CDs do not apply either to leaf nodes in the goal tree, or to the whole tree, but to subtrees" [14].

Subsetability clearly does not describe low-level perceptual processes. And, by definition, subsetability does not describe large cognitive tasks. Instead, it describes how a large cognitive task (e.g. learn a PL/E) can be decomposed into smaller, manageable cognitive tasks (e.g. learn a sequence of PL/E subsets).

7.6 Effect of Manipulation

"It ought to be possible to consider each dimension and say 'if you change the design in the following way, you will move its value on this dimension'. That's a criterion of understanding how the dimension works, as well as the basis for design manoeuvres... So the criterion is that when we define a new dimension, we should be able to say something about how to manipulate it" [14].

Consider this design maneuver: to increase subsetability, add defaults to the notation and/or environment. Defaults can hide features of the notation and/or environment, allowing them to appear in lower (i.e. more complex) microworlds in the lattice. For example, the Java notation has a "package" feature, as well as a default package. The existence of the default package shields the user from the package feature when working within higher (i.e. simpler) microworlds in the lattice. The cost of adding defaults is decreased visibility: the features of the notation and/or environment that have been replaced by defaults will no longer be visible to the user. So the user might not know of their existence.

Clearly subsetability can be manipulated via defaults.

8 Conclusion

In this paper we defined the concept of subsetability. A world is subsetable to the extent that it can be decomposed into a lattice of microworlds, where each microworld consists of an equipment set, a physical setting, and a task specification. More specifically, we argued that subsetability applies to the world of notational systems, and that subsetability may be a new cognitive dimension of notational systems. Even more specifically, we argued that subsetability applies to notational system of programming languages and environments (PL/Es). Furthermore, we described some theoretical research which strongly argues that a world's subsetability positively affects its learnability and teachability. We also noted that, sadly, little empirical support for that argument exists.

In doing so, we described connections between subsetability and several largely disjoint threads of research:

- The theoretical observations of Simon regarding the learnability and teachability of nearly decomposable hierarchies.

- The theoretical observations of Papert, Burton, Brown, Fischer, and others regarding the value of microworlds and sequences of increasingly complex microworlds as an approach to learning and teaching.
- The pragmatic research of Findler, Brusilovsky, DePasquale, and others on the value of PL/E subsets.
- The theoretical yet pragmatic work of Green, Blackwell, and others on cognitive dimensions of notational systems.

Clearly more empirical research on subsetability is needed. If the empirical research supports the theoretical arguments, then we believe that subsetability will be prescriptive for programming education. The concept will prescribe how learners can best approach the learning of a PL/E, and how teachers can best provide support for that learning. We expect that software engineering students, as well as non-professional end-user programmers, will benefit from subsetability.

Moreover, we believe that subsetability will be prescriptive for software engineering itself. Subsetability prescribes how one should learn and teach a PL/E. But when "stood on its head," is also prescribes how one should design a PL/E so it is easy to learn and teach. Certainly PL/E designers have many goals of varying priorities; learnability and teachability may not be high on the list. Nevertheless, a PL/E that is easy to learn and teach has a greater chance of acceptance than one that is not. So PL/E designers cannot afford to ignore subsetability.

Acknowledgments

This work was supported in part by the EUSES Consortium via NSF grant CCR-0324844.

References

1. Green, T. and A. Blackwell, *Cognitive Dimensions of Information Artefacts: A Tutorial (Version 1.2 October 1998)*. 1998.
2. Simon, H.A., *The Sciences of the Artificial*. Third ed. 2001, Cambridge, MA: The MIT Press.
3. Papert, S., *Mindstorms: Children, Computers, and Powerful Ideas*. 1980, New York: Basic Books.
4. Burton, R.R., J.S. Brown, and G. Fischer, *Skiing as a Model of Instruction*, in *Everyday Cognition: Its Development in Social Context*, B. Rogoff and J. Lave, Editors. 1984, Harvard University Press: Cambridge, MA. p. 139-150.
5. Gray, K.E. and M. Flatt. *ProfessorJ: A Gradual Introduction to Java through Language Levels*. 2003. OOPSLA'03.
6. Findler, R.B., et al. *DrScheme: A Pedagogic Programming Environment for Scheme*. In *International Symposium on Programmming Languages, Implementations, Logics, and Programs*. 1997.
7. Findler, R.B., et al., *DrScheme: A Programming Environment for Scheme.* Journal of Functional Programming, 2002. **12**(2): p. 159-182.

8. Brusilovsky, P., et al. *Teaching Programming to Novices: A Review of Approaches and Tools*. In *ED_MEDIA'94: World Conference on Educational Multimedia and Hypermedia*. 1994.
9. Brusilovsky, P., et al., *Mini-languages: A Way to Learn Programming Principles.* Education and Information Technologies, 1997. **2**(1): p. 65-83.
10. Kelleher, C. and R. Pausch, *Lowering the Barriers to Programming: A Taxonomy of Environments and Languages for Novice Programmers.* ACM Computing Surveys, 2006. **37**(2): p. 83-137.
11. Catrambone, R. and J.R. Carroll. *Learning a Word Processing System with Training Wheels and Guided Exploration*. In *SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface CHI '87*. 1987. New York: ACM Press.
12. Puntambekar, S. and R. Hübscher, *Tools for Scaffolding Students in a Complex Learning Environment: What have we Gained and What have we Missed?* Educational Psychologist, 2005. **40**(1): p. 1-12.
13. DePasquale, P., J.A.N. Lee, and M.A. Pérez-Quiñones. *Evaluation of Subsetting Programming Language Elements in a Novice's Programming Environment*. In *Proceedings of SIGCSE'04*. 2004. New York: ACM Press.
14. Blackwell, A.F. *Dealing with New Cognitive Dimensions*. In *Workshop on Cognitive Dimensions*. 2000. University of Hertfordshire.